Flattening Traits

Oscar Nierstrasz, Stéphane Ducasse and Nathanael Schärli

Institut für Informatik und Angewandte Mathematik
University of Bern, Switzerland

IAM-05-005

April, 2005

Abstract

Traits are fine-grained components that can be used to compose classes, while avoiding many of the problems of multiple inheritance and mixin-based approaches. Since most implementations of traits have focused on dynamically-typed languages, the question naturally arises, how can one best introduce traits to statically-typed languages, like Java and C#? In this paper we argue that the *flattening property* of traits should be used as a guiding principle for any attempt to add traits to statically-typed languages. This property essentially states that, semantically, *traits can be compiled away.* We demonstrate how this principle applies to Featherweight-Trait Java, a conservative extension to Featherweight Java.

CR Categories and Subject Descriptors: D.3.1 [Programming Languages]: Formal Definitions and Theory—*Semantics*; F.3.3 [Logics and Meanings of Programs]: Studies of Program Constructs—*Type structure*; F.4.1 [Mathematical Logic and Formal Languages]: Mathematical Logic—*Lambda calculus and related systems*

1 Introduction

Traits were introduced [16, 17] as a simple programming language mechanism for incrementally composing classes from small, reusable components, while avoiding problems of fragility in the class hierarchy that arise with approaches based on mixins or multiple inheritance. Initial experiences using traits in SMALLTALK to refactor complex class hierarchies have been very promising [2], and the question naturally arises, how can we apply traits to statically-typed languages like Java and C#?

Traits are essentially sets of methods, divorced from any instance variables or a superclass. Composite traits may be composed from subtraits using the trait composition operators, *sum*, *aliasing* and *exclusion*. A trait is bound to specific instance variables and a superclass only when that trait is used in the composition of a given class. A trait is consequently very much like an abstract class, so perhaps traits in statically-typed languages should be treated the same way that abstract classes are. In particular, this would typically mean that every named trait will define a type, since classes in C++, Java and C# define types.

The flaw in this reasoning is that traits support the *flattening property*, which says that the semantics of a method defined in a trait is identical to the semantics of the same method defined in a class that uses the trait. In principle, then, *traits can be compiled away*. But if traits can be compiled away, then what happens to the types that they define?

We propose that the flattening property actually provides us with a principle for answering this and other questions. Instead of first asking how to integrate traits with the semantics of a given language, we should answer the question, how can we flatten traits to the base language. Once we know how to flatten traits, we will know how to extend the language, since the design space will then be drastically reduced.

In particular, if we are interested in adding traits to a statically-typed language $\mathcal{L}$, then a program p should be type-safe in the extended language $\mathcal{T}$ if and only if the flattened program $[\![p]\!]$ is type-safe in $\mathcal{L}$.

FEATHERWEIGHT JAVA (FJ) is an object calculus that captures just those aspects of Java that are needed to explore certain questions concerning Java's type system [8, 9]. In particular, FJ was originally used to ascertain that Java's type system could be extended to accommodate generics without breaking existing programs. Since traits also offer a conservative extension to Java-like languages and exhibit certain aspects of genericity, a natural starting point for applying traits to Java-like languages (including C#) would be an investigation of introducing traits to FJ. Liquori and Spiwack have taken FJ as a starting point to define FEATHERWEIGHT-TRAIT JAVA(FTJ) [11], a conservative extension of FJ that adds statically typed traits.

We provide a brief overview of traits in Section 2. In Section 3 we show how programs in FTJ can be flattened to FJ. This allows us to apply an "acid test" to FTJ— expressions in FTJ should be type-safe if and only if their flattened counterparts are type-safe in FJ. We show that, with some small caveats, this is in fact the case. In Section 4 we take the same approach to investigate how named traits can be used to stand for types, if we first extend our base language to support interfaces. In Section 5 we investigate the introduction of traits in FGJ (FJ extended with generics). The principle of flattening leads us naturally to a notion of *generic traits*. In Section 6 we provide a brief overview how traits are implemented in Smalltalk. We briefly survey some related work in Section 7. We conclude in Section 8 with some remarks about ongoing and future work.

2 Traits in a Nutshell

Traits [17] are essentially groups of methods that serve as building blocks for classes and are primitive units of code reuse. As such, they allow one to factor out common behavior and form an intermediate level of abstraction between single methods and complete classes. A trait consists of *provided methods* that implement its behavior, and of *required methods* that parameterize the provided behavior. Traits cannot specify any instance variables, and the methods provided by traits never directly access instance variables. Instead, required methods can be mapped to state when the trait is used by a class.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* that are implemented at the level of the class. These glue methods connect the traits together and can serve as accessor for the necessary state. The semantics of such a class is defined by the following three rules:

- *Class methods take precedence over trait methods.* This allows the glue methods defined in the class to override equally named methods provided by the traits.
- *Flattening property.* A non-overridden method in a trait has the same semantics as the same method implemented in the class.
- *Composition order is irrelevant.* All the traits have the same precedence, and hence conflicting trait methods must be explicitly disambiguated.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. Traits enforce explicit resolution of conflicts by implementing a glue method at the level of the class that overrides the conflicting methods, or by *method exclusion*, which allows one to exclude the conflicting method from all but one trait. In addition traits allow *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable, for example, because it has been overridden.

Example: Geometric Objects. Suppose we want to represent a graphical object such as a circle or square that is drawn on a canvas. Such a graphical object can be decomposed into three reusable aspects — its geometry, its color and the way that it is drawn on a canvas.

Figure 1 shows this for the case of a Circle class composed from traits TCircle, TColor and TDrawing:

- TCircle defines the geometry of a circle: it requires the methods center, center:, radius, and radius: and provides methods such as bounds, hash, and =.
- TDrawing requires the methods drawOn: bounds and provides the methods draw, refresh, and refreshOn:.
- TColor requires the methods rgb, rgb: and provides all kind of methods manipulating colors. We only show the methods hash and = as they will conflict with others at composition time.

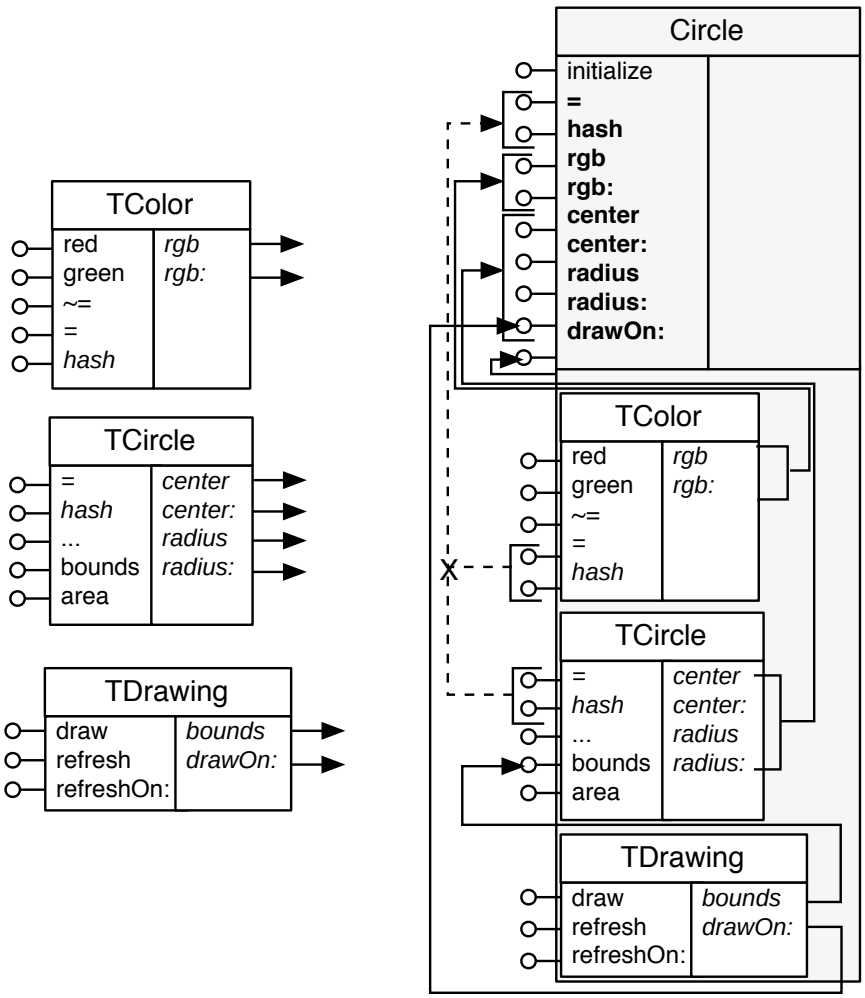

Figure 1: Class Circle is composed from traits TCircle, TColor and TDrawing.

CL	::=	class C ◁ C {$\overline{\text{C}}$ $\overline{\text{f}}$;K $\overline{\text{M}}$ $\overline{\text{TA}}$}	*Classes*
TL	::=	trait T is {$\overline{\text{M}}$; $\overline{\text{TA}}$}	*Traits*
TA	::=	T | TA with {m@n} | TA minus {m}	*Trait expressions*
K	::=	C($\overline{\text{C}}$ $\overline{\text{f}}$) {super($\overline{\text{f}}$);this.$\overline{\text{f}}$=$\overline{\text{f}}$;}	*Constructors*
M	::=	C m($\overline{\text{C}}$ $\overline{\text{x}}$) {↑e;}	*Methods*
e	::=	x | e.f | e.m($\overline{\text{e}}$) | new C($\overline{\text{e}}$) | (C)e	*Expressions*

Figure 2: FTJ Syntax.

The class Circle specifies three instance variables center, radius, and rgb and their respective accessor methods. It is composed from the three traits TDrawing, TCircle, and TColor. As there is a conflict for the methods hash and = between the traits TCircle and TColor, we alias those methods in both traits to be able to access them in the methods hash and = of the class Circle resolving the conflicts.

3 Flattening traits

FJ strips Java down to a tiny functional calculus that expresses just enough of the language to reason about the essential type features of Java. Issues like side effects, concurrency and reflection are forgotten, but even some type issues such as interfaces and overloading are left out as being non-essential.

Liquori and Spiwack have defined FTJ as a conservative extension of FJ, with minimal syntactic and semantic changes to accommodate traits. But is their interpretation of traits reasonable?

To answer this question, we explore in this section an alternative approach to defining FTJ by *flattening*, *i.e.*, by translation to FJ. In other words, we compile traits away to obtain pure FJ programs. We then show that the static and dynamic semantics of FTJ programs is (largely) consistent with that of the flattened programs in FJ.

The point of this exercise is to provide evidence that FTJ is in fact a reasonable extension of FJ to traits, precisely because it correctly interprets the flattening property. In general, we argue, any type system that accommodates traits should have the property that programs with traits should be equivalent, in some way, to their flattened counterparts in the base language.

3.1 Featherweight Trait Java

The syntax of FTJ is shown in Figure 2. The only differences with the syntax of FJ are the modification of class definitions to include a sequence of *used traits* $\overline{\text{TA}}$, and the addition of syntax for trait definitions (TL) and trait expressions (TA). As in FJ, the notation $\overline{\text{C}}$ denotes a possible empty sequence of elements C (with or without commas, as appropriate; • represents the empty sequence.) For the sake of conciseness we abbreviate the keyword `extends` to the symbol ◁ and the keyword `return` to the symbol ↑.

With traits, the behavior of a class is specified as the composition of traits and some *glue methods* ($\overline{\text{M}}$) that are implemented at the level of the class (CL) or the composite trait (TL). These glue methods connect the traits together and can serve as accessor for the necessary state.

The operational semantics of FTJ specifies a modified method lookup algorithm that ensures that methods of a class `C` take precedence over methods provided by any of the used traits $\overline{\texttt{TA}}$. Similarly, methods of a named trait `T` take precedence over methods provided by subtraits $\overline{\texttt{TA}}$ used by `T`.

Because the composition order is irrelevant, a *conflict* arises if we combine two or more traits that provide identically named methods that do not originate from the same trait. $\overline{\texttt{TA}}$ is a composition of traits $\texttt{T}_i$, possibly giving rise to conflicts. Conflicts may be resolved by overriding them with glue methods $\overline{\texttt{M}}$ in the class using $\overline{\texttt{TA}}$, or by excluding the conflicting methods. `TA minus {m}` removes the method named `m` from the trait expression `TA`.

In addition traits allow *method aliasing*. The programmer can introduce an additional name for a method provided by a trait to obtain access to a method that would otherwise be unreachable because it has been overridden. `TA with {m@n}` defines `m` to be an alias for the existing method named `n`. (Note that the aliasing syntax of FTJ (`m@n`) puts the new name `n` *after* the existing method name `m`, whereas the aliasing operator ($\rightarrow$) expects its arguments in the reverse order.)

3.2 Flattening FTJ

We have previously developed a simple set-theoretic model of traits [15]. The goals of this model were to define the trait composition operators, to give an operational account of method lookup (particularly `self`- and `super`-sends), and to develop a notion of equivalence for traits. The model further makes precise the notion of method *conflicts* arising during trait composition, and the notion that a class constructed using traits can always be flattened into one that does not use traits.

The trait model defines *method dictionaries* as mappings from method names to method bodies. A *trait* is just a method dictionary in which some method names may be bound instead to $\top$, representing a conflict. Traits may be constructed using the operators $+$ (composition), $-$ (exclusion), $\triangleright$ (overriding) and $[\rightarrow]$ (aliasing). The key point is that traits are always composed using the composition operator $+$, which is associative and commutative [6], hence insensitive to the order in which traits are composed. Conflicts are resolved by the composing class by overriding or excluding the conflicts [17]. We shall use this framework for flattening FTJ.

The flattening property simply states that we can always evaluate the trait composition operators occurring within a class definition to obtain an equivalent class whose method dictionary does not refer to traits — that is, the traits can be compiled away. In order to flatten FTJ programs, then, we must interpret the parts of the FTJ syntax that represent method dictionaries and traits, and we must define the trait composition operators for those syntactic entities. The translation from FTJ to FJ will simply evaluate the composition operators.

Figure 3 presents the trait composition operators interpreted in the context of FTJ. These operators are used to define the flattening function $[\![\cdot]\!]$ which translates an FTJ class to an FJ class in Figure 4.

We interpret a sequence of methods $\overline{\texttt{M}}$ as representing a method dictionary, and sequence of trait expressions $\overline{\texttt{TA}}$ as representing a trait composition $\sum_i \texttt{TA}_i$

In order to define the composition operators, we first need a couple of auxiliary functions. *lookup*$(\texttt{m}, \overline{\texttt{M}})$ (1) returns the declaration of method `m` in $\overline{\texttt{M}}$, if present. $\perp$ represents an undefined method. *extract*$(X, \overline{\texttt{M}})$ (2) returns the subsequence of $\overline{\texttt{M}}$ containing the definitions of the

$$lookup(\mathtt{m}, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \begin{cases} M & \text{if } M = \mathtt{C\ m(}\overline{\mathtt{C}}\ \mathtt{x)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \bot & \text{otherwise} \end{cases} \quad (1)$$

$$extract(X, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \bigwedge_{\mathtt{m} \in X} lookup(\mathtt{m}, \overline{\mathtt{M}}) \quad (2)$$

$$mNames(\overline{\mathtt{M}}) \stackrel{\text{def}}{=} \{\mathtt{m} \mid lookup(\mathtt{m}, \overline{\mathtt{M}}) \neq \bot\} \quad (3)$$

$$\frac{\mathtt{trait\ T\ is\ \{}\overline{\mathtt{M}}\mathtt{;}\overline{\mathtt{TA}}\mathtt{\}}}{local(\mathtt{T}) = \overline{\mathtt{M}}} \qquad \frac{\mathtt{trait\ T\ is\ \{}\overline{\mathtt{M}}\mathtt{;}\overline{\mathtt{TA}}\mathtt{\}}}{subtraits(\mathtt{T}) = \overline{\mathtt{TA}}} \quad (4)$$

$$\overline{\mathtt{M}} - \mathtt{m} \stackrel{\text{def}}{=} \overline{\mathtt{M}} \backslash lookup(\mathtt{m}, \overline{\mathtt{M}}) \quad (5)$$

$$\overline{\mathtt{M}}_1 \rhd \overline{\mathtt{M}}_2 \stackrel{\text{def}}{=} \overline{\mathtt{M}}_1, (\overline{\mathtt{M}}_2 \backslash extract(mNames(\overline{\mathtt{M}}_1), \overline{\mathtt{M}}_2)) \quad (6)$$

$$\overline{\mathtt{M}}[\mathtt{n}{\rightarrow}\mathtt{m}] \stackrel{\text{def}}{=} \begin{cases} (\overline{\mathtt{M}} \backslash lookup(\mathtt{n}, \overline{\mathtt{M}})), conflict(\mathtt{n}) & \text{if } lookup(\mathtt{n}, \overline{\mathtt{M}}) \neq \bot \\ \overline{\mathtt{M}}, \mathtt{C\ n(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e;\}} & \text{else if } \mathtt{C\ m(}\overline{\mathtt{C}}\ \overline{\mathtt{x}}\mathtt{)\{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \overline{\mathtt{M}} & \text{otherwise} \end{cases} \quad (7)$$

$$mBodies(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2) \stackrel{\text{def}}{=} extract(mNames(\overline{\mathtt{M}}_1) \backslash mNames(\overline{\mathtt{M}}_2), \overline{\mathtt{M}}_1) \quad (8)$$

$$broken(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2) \stackrel{\text{def}}{=} (mNames(\overline{\mathtt{M}}_1) \cap mNames(\overline{\mathtt{M}}_2)) \backslash mNames(\overline{\mathtt{M}}_1 \cap \overline{\mathtt{M}}_2) \quad (9)$$

$$\overline{\mathtt{M}}_1 + \overline{\mathtt{M}}_2 \stackrel{\text{def}}{=} mBodies(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2), mBodies(\overline{\mathtt{M}}_2, \overline{\mathtt{M}}_1), (\overline{\mathtt{M}}_1 \cap \overline{\mathtt{M}}_2), \bigwedge\{conflict(\mathtt{m}) \mid \mathtt{m} \in broken(\overline{\mathtt{M}}_1, \overline{\mathtt{M}}_2)\} \quad (10)$$

$$\text{where } conflict(\mathtt{m}) = \mathtt{Object\ m()\ \{}\uparrow\bot\mathtt{;\ \}}$$

Figure 3: Composition operators for FTJ

$$[\![\mathtt{class\ C} \lhd \mathtt{D\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;K}\ \overline{\mathtt{M}}\ \overline{\mathtt{TA}}\mathtt{\}}]\!] \stackrel{\text{def}}{=} \mathtt{class\ C} \lhd \mathtt{D\ \{}\overline{\mathtt{C}}\ \overline{\mathtt{f}}\mathtt{;K}\ \overline{\mathtt{M}} \rhd [\![\overline{\mathtt{TA}}]\!]\mathtt{\}} \quad (11)$$

$$[\![\overline{\mathtt{TA}}]\!] \stackrel{\text{def}}{=} \sum_{\mathtt{TA}_i \in \overline{\mathtt{TA}}} [\![\mathtt{TA}_i]\!] \quad (12)$$

$$[\![\mathtt{T}]\!] \stackrel{\text{def}}{=} local(\mathtt{T}) \rhd [\![subtraits(\mathtt{T})]\!] \quad (13)$$

$$[\![\mathtt{TA\ with\ m@n}]\!] \stackrel{\text{def}}{=} [\![\mathtt{TA}]\!][n{\rightarrow}m] \quad (14)$$

$$[\![\mathtt{TA\ minus\ m}]\!] \stackrel{\text{def}}{=} [\![\mathtt{TA}]\!] - \mathtt{m} \quad (15)$$

Figure 4: Flattening FTJ to FJ

methods named in X (where $\bigwedge$ builds a sequence from its operands — if X is empty, then *extract* returns $\bullet$, the empty sequence). *mNames*($\overline{\texttt{M}}$) (3) returns the set of method names of methods declared in $\overline{\texttt{M}}$. We will also make use of *local*(T) and *subtraits*(T) (4), which return, respectively, the methods and the subtraits of a named trait T.

The exclusion operator (5) simply removes[1] the definition of m from the method dictionary $\overline{\texttt{M}}$. Overriding (6) removes from $\overline{\texttt{M}}_2$ those methods already defined in $\overline{\texttt{M}}_1$, and concatenates what remains to $\overline{\texttt{M}}_1$. Aliasing (7) simply concatenates an existing method definition for m under the new name n. If, however, the "new" name n is already bound in $\overline{\texttt{M}}$, then a conflict is generated instead. (If m is absent, then we can just ignore the alias, so that any references to n will generate errors.) Note that we have chosen here to represent a conflict by the method body {↑⊥;}. The flattening function will therefore yield a valid FJ program if and only if all conflicts are resolved. (An alternative approach could be to generate FJ code that is syntactically valid, but contains a type error, such as a call to a non-existent method.)

Trait composition is slightly more complicated to define. We first define the auxiliary functions *mBodies* and *broken*. *mBodies*($\overline{\texttt{M}}_1, \overline{\texttt{M}}_2$) (8) represents the method declarations in $\overline{\texttt{M}}_1$ that do not conflict with any methods declared in $\overline{\texttt{M}}_2$. $\overline{\texttt{M}}_1 \cap \overline{\texttt{M}}_2$ represents the method declarations that are (syntactically) *identical* in $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$ (once again abusing set notation to represent intersection of the method dictionaries). These methods also do not pose any conflicts. *broken*($\overline{\texttt{M}}_1, \overline{\texttt{M}}_2$) (9) represents the set of names of methods with non-identical declarations in both $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$. These represent actual conflicts. Finally, the composition of $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$ (10) concatenates the non-conflicting and conflicting method declarations.

Now we are ready to define the translation function $[\![\cdot]\!]$ (Figure 4). A flattened class is one in which its locally defined methods override the (flattened) methods of the used traits (11). Flattening a sequence of FTJ traits or a trait expression always yields a (possibly empty) sequence of FJ methods. A sequence of traits (12) translates to the composition of the translation of its parts. The local methods of a named trait (13) override the composition of its subtraits. Aliasing (14) and exclusion (15) are directly interpreted by the aliasing and exclusion operators.

3.3 Equivalence of trait-based and flattened programs

Now we can attempt to answer the question, does FTJ provide a reasonable interpretation of traits?

Ideally, we expect the following result to hold:

> If e is an expression in an FTJ program P, then e is well-typed in FTJ if and only if $[\![\texttt{e}]\!]$ is well-typed in FJ.

As it turns out, we can obtain a very similar result, but due to some minor differences in the interpretation of traits in the two approaches (*i.e.*, the FTJ type system and our flattening approach), we must state a slightly weaker result.

FTJ follows the formal trait model [15] fairly closely, but there are a number of small discrepancies.

[1] Note that we also adopt the convention initiated by Igarashi *et al.* [9] of using set-based notation for operators over sequences: $M = \texttt{C m(}\overline{\texttt{C}}\ \texttt{x)}\ \ldots \in \overline{\texttt{M}}$ means that the method declaration M occurs in $\overline{\texttt{M}}$, whereas $\overline{\texttt{M}} \backslash M$ stands for the sequence $\overline{\texttt{M}}$ with M removed. $\overline{\texttt{M}}_1, \overline{\texttt{M}}_2$ is the concatenation of the sequences $\overline{\texttt{M}}_1$ and $\overline{\texttt{M}}_2$. This abuse of notation is justified since the order in which the elements occur in $\overline{\texttt{M}}$ is irrelevant.

1. First, the aliasing mechanism of FTJ is more ambitious, automatically converting recursive calls to the new aliased name. So, if method `m` contains the expression `this.m(e)`, and `m` is aliased to `m1`, then the body of `m1` will be rewritten to contain `this.m1(e)`. This innovation is not part of the original definition of the trait model, and is not reflected in the flattening.

2. Next, in the case where one attempts to define an alias `m1` when `m1` is already declared, the trait model specifies that a conflict be generated (which is what the flattening function does). FTJ, on the other hand, deals with this not operationally, but prohibits this kind of aliasing at the level of the type system. This means the FTJ is more restrictive, since such conflicts cannot be repaired by means of glue code. One can debate which interpretation is preferable.

3. Finally, FTJ adopts the principle that two methods with the same name don't conflict only if they originate from the same subtrait. This is perfectly consistent with the implementation of traits in Squeak [17]. The formal trait model leaves this point open, however, allowing different interpretations of when two methods are "the same". In the concrete case of our flattening function, the definitions of *broken* (9) and + (10) make use of the construct $\overline{\mathtt{M}}_1 \cap \overline{\mathtt{M}}_2$ to assess which methods are the same. Since methods in FJ are just syntactic entities, this means conflicts only arise in flattening when two methods are syntactically different. So flattening is more liberal than the FTJ type system.

The first point interferes with our ideal result, so we must exclude such programs. The second point poses no problems, since FTJ's type system will reject programs with invalid aliasing. The third point breaks the two-way implication: if we have traits `TA` and `TB` that both provide a method `m` with syntactically identical implementations, and a class `C` uses both `TA` and `TB` without overriding `m`, then the FTJ type system will flag this as a conflict, whereas our flattening function will simply unify the two methods.

As a consequence, the best we can hope for is the following:

> If `e` is well-typed within an FTJ program P, in which recursive methods are not aliased, then `e` is also well-typed in the FJ program $[\![P]\!]$.

In order to obtain the stronger result, we should adapt either FTJ or our flattening function to deal consistently with "not nice" programs. For example, we could modify our flattening function to accommodate the FTJ interpretation of aliasing (7) by generating `{`$\uparrow$`e[this.n/this.m]}` as the body of the new method `n`.

We should also adapt the flattening function to be consistent with the more restrictive interpretation of when methods conflict. A trivial solution would be to decorate methods originating from traits with the name of the defining trait. We would then compare sets of tuples $\{(\mathtt{T}_i, \mathtt{M}_i)\}$ rather than simply sets of methods $\overline{\mathtt{M}}$ in the definitions of *broken* and +.

Given our partial result, we can conclude that FTJ indeed offers a reasonable interpretation of traits that is consistent with the flattening property.

4 Traits, types and interfaces

As should be evident from the syntax of FTJ alone, traits in FTJ do not define types. And because FJ and FTJ do not model interfaces, this means that only class names may be used

to specify the signature of a method. While this simplifies the theoretical foundation of these models, it poses serious practical problems because it makes it hard or impossible to write traits that can be used across multiple classes.

As an example, suppose that we would like to have a trait TRectangle that is used to build two classes Rectangle and VisualRectangle, which have Object as their only common superclass. This trait should provide, amongst others, a method includes, which takes another rectangle as an argument and returns a boolean indicating whether the argument rectangle is fully included in the receiver. While the method includes can conceptually take as its argument an object of any class that uses the trait TRectangle (*e.g.*, Rectangle and VisualRectangle), FTJ does not allow us to express this since trait names are not valid types.

One way to avoid this problem would be to extend FTJ so that traits, like classes, also define types. In the above example, this means that the trait TRectangle will also define a corresponding type with the same name that can then be used to define the type of the argument in the signature of the method includes. However, in order for this to work, we also need to extend the definition of subtyping in FTJ so that each class that uses the trait TRectangle is a subtype of the type that is implicitly defined by this trait. And since we want to flatten FTJ programs to FJ, this means that we need to add this form of multiple subtyping also to FJ.

Since we need to extend FJ with a form of multiple subtyping anyway, an alternative approach would be to introduce the notion of interfaces into the calculus. This means that as in regular Java, each interface defines an FJ type, and classes as well as traits can be declared to be subtypes of numerous interface types. Even though traits themselves cannot be used as types, this allows us to solve the identified problem because we can declare a corresponding interface for each trait that should be used as a type. In our example, this means that we declare an interface IRectangle containing the same method signatures as the trait TRectangle, and that we then declare all "rectangle-like" classes (in particular all classes that use the trait TRectangle) as subtypes of IRectangle by implementing this interface.

While both approaches, introducing interfaces or using traits as types, require adding multiple subtyping to the calculi, there are important conceptual differences between these two approaches. While the approach of treating each trait as a type may be more convenient in practice, the presence of exclusions and aliases add a certain complexity to the subtype relation. Furthermore, making each trait be a type blurs the important conceptual distinction between implementation and interfaces, which leads to two kind of problems. First, it does not address the fact that in the same way as subclassing does not necessarily imply subtyping [4], a trait may be composed form another trait without conceptually being a subtype of it. Second, if there are multiple traits providing different implementations of the same conceptual interface (*e.g.*, TRectangle and TOptimizedRectangle), we end up with multiple identical types.

All these problems are avoided if we do not consider traits as types and use interfaces instead. However, this comes at the cost that whenever traits are composed, the necessary interfaces and subtype relationships have to be explicitly declared. Note that this is this approach that has been followed by Denier and Cointe in their implementation of traits with AspectJ [5].

4.1 FJI and FTJI

We now explore an approach in which traits generate interfaces rather than types. We will first extend FJ with interfaces, obtaining Featherweight Java with Interfaces (FJI).

CL	::=	class C ◁ C implements $\overline{\texttt{I}}$ {$\overline{\texttt{S}}$ $\overline{\texttt{f}}$;K $\overline{\texttt{M}}$}	*Classes*
ID	::=	interface I ◁ $\overline{\texttt{I}}$ {$\overline{\texttt{SG}}$}	*Interfaces*
S	::=	C | I	*Types*
SG	::=	S m($\overline{\texttt{S}}$)	*Method signatures*
K	::=	C($\overline{\texttt{S}}$ $\overline{\texttt{f}}$) {super($\overline{\texttt{f}}$);this.$\overline{\texttt{f}}$=$\overline{\texttt{f}}$;}	*Constructors*
M	::=	S m($\overline{\texttt{S}}$ $\overline{\texttt{x}}$) {↑e;}	*Methods*
ID	::=	interface I ◁ $\overline{\texttt{I}}$ {$\overline{\texttt{SG}}$}	*Interfaces*
e	::=	x | e.f | e.m($\overline{\texttt{e}}$) | new C($\overline{\texttt{e}}$) | (S)e	*Expressions*

Figure 5: FJI Syntax.

$$\frac{}{\texttt{S}<:\texttt{S}} \qquad \frac{\texttt{class C} \lhd \texttt{D implements } \overline{\texttt{I}} \texttt{ \{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K}\ \overline{\texttt{M}}\texttt{\}}}{\texttt{C}<:\texttt{D} \quad \forall i.\texttt{C}<:\texttt{I}_i}$$

$$\frac{\texttt{S}<:\texttt{S}' \quad \texttt{S}'<:\texttt{S}''}{\texttt{S}<:\texttt{S}''} \qquad \frac{\texttt{interface I} \lhd \overline{\texttt{I}} \texttt{ \{}\overline{\texttt{SG}}\texttt{\}}}{\forall i.\texttt{I}<:\texttt{I}_i}$$

Figure 6: FJI Subtyping.

Then we define Featherweight-Trait Java with Interfaces (FTJI) as an extension of FTJ.

In fact, FJI is rather trivial to define. Figure 5 shows the syntax of FJI. The semantics of FJI is almost identical to that of FJ. The rules for *Small-step operational semantics* and *Congruence* are unchanged. The rules for *Field lookup*, *Method body lookup*, *Expression typing* and *Class typing* require only trivial changes to reflect the new syntax for classes and types. Finally, the rules for *Subtyping*, *Method type lookup* and *Method typing* require straightforward extensions to accommodate the fact that interface definitions introduce new types. As an example, we show the new subtyping rules for FJI in Figure 6.

We show a possible syntax for FTJI in Figure 7. Traits are as before in FTJ, with one important difference: method signatures may now refer to trait names, since types may be class names, interface names or trait names.

What does this imply for flattening? Clearly the only sensible approach is to translate traits to interfaces — every trait declaration will generate an interface declaration in the flattened system. We present a possible way of flattening FTJI to FJI in Figure 8.

We flatten classes as before, expanding the methods of all used traits. However we additionally generate an interface for every declared trait name, so that trait names may be used

CL	::=	class C ◁ C implements $\overline{\texttt{I}}$ {$\overline{\texttt{S}}$ $\overline{\texttt{f}}$;K $\overline{\texttt{M}}$ $\overline{\texttt{TA}}$}	*Classes*
S	::=	C | I | T	*Types*

TL and TA are as in Figure 2 and ID, SG, K, M, ID, and e are as in Figure 5.

Figure 7: FTJI Syntax.

$$[\![\texttt{class C} \lhd \texttt{D implements } \overline{\texttt{I}}\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}}\ \overline{\texttt{TA}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \texttt{class C} \lhd \texttt{D implements } \overline{\texttt{I}}\ \mathit{subtraitNames}(\overline{\texttt{TA}})\ \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K } \overline{\texttt{M}} \rhd [\![\overline{\texttt{TA}}]\!]\texttt{\}} \quad (16)$$

$$[\![\texttt{trait T is \{}\overline{\texttt{M}}\texttt{;}\overline{\texttt{TA}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \texttt{interface T} \lhd \mathit{subtraitNames}(\overline{\texttt{TA}})\ \texttt{\{}\mathit{interface}(\overline{\texttt{M}})\,\mathit{aliases}(\overline{\texttt{TA}})\texttt{\}} \quad (17)$$

$$\mathit{subtraitNames}(\overline{\texttt{TA}}) \stackrel{\text{def}}{=} \bigwedge_i \mathit{subtraitNames}(\texttt{TA}_i) \quad (18)$$

$$\mathit{subtraitNames}(\texttt{T}) \stackrel{\text{def}}{=} \texttt{T}$$

$$\mathit{subtraitNames}(\texttt{TA with \{m@n\}}) \stackrel{\text{def}}{=} \mathit{subtraitNames}(\texttt{TA})$$

$$\mathit{subtraitNames}(\texttt{TA minus \{m\}}) \stackrel{\text{def}}{=} \mathit{subtraitNames}(\texttt{TA})$$

$$\mathit{interface}(\overline{\texttt{M}}) \stackrel{\text{def}}{=} \bigwedge_i \mathit{interface}(\texttt{M}_i) \quad (19)$$

$$\mathit{interface}(\texttt{S m(}\overline{\texttt{S}}\ \overline{\texttt{x}}\texttt{) \{}\uparrow\texttt{e;\}}) \stackrel{\text{def}}{=} \texttt{S m(}\overline{\texttt{S}}\texttt{)}$$

$$\mathit{aliases}(\overline{\texttt{TA}}) \stackrel{\text{def}}{=} \bigwedge_i \mathit{aliases}(\texttt{TA}_i) \quad (20)$$

$$\mathit{aliases}(\texttt{T}) \stackrel{\text{def}}{=} \bullet$$

$$\mathit{aliases}(\texttt{TA with \{m@n\}}) \stackrel{\text{def}}{=} \texttt{S n(}\overline{\texttt{S}}\texttt{)},\ \mathit{if}\ \texttt{S m(}\overline{\texttt{S}}\texttt{) \{...\}} \in \mathit{lookup}(\texttt{m}, [\![\texttt{TA}]\!])$$

$$\mathit{aliases}(\texttt{TA minus \{m\}}) \stackrel{\text{def}}{=} \bullet$$

The translation of $\overline{\texttt{TA}}$ is the same as in Figure 4.

Figure 8: A possible flattening of FTJI to FJI

as types. The only wrinkle in this translation is what to do about exclusion. If a trait `TA` uses a trait `TB`, but excludes a method `m` from `TB`, then it is clear that the interface `TA` no longer properly extends `TB`. However, exclusion is mainly intended as a mechanism to resolve conflicts, not for "editing" traits. With this principle in mind, we should assume that `TA` *will* in fact be a proper extension of `TB` since the method `m` that is excluded is implemented in `TA` by some other path. The flattening function we present in Figure 8 takes this approach, since *subtraitNames* extracts all used trait names, including those for which some methods have been excluding. We rely on the fact that the type system of FJI will complain if the resulting interface declarations lead to an inconsistency.

In FTJI, we can now declare the trait TRectangle to contain a method expecting an argument of type TRectangle. Flattening to FJI tells us that this trait should be treated as if it were an interface. Any class that uses this trait will then automatically implement the interface TRectangle.

5 Generic traits

While multiple subtyping allows us to define the signature of the method includes so that it is not specific to a single class, FTJ still suffers from a lack of expressiveness when it comes to defining reusable trait methods. As an illustration, assume that the trait TRectangle also provides a binary method intersect:, which takes another rectangle as an argument and returns a new rectangle that is the intersection between the receiver and the argument. If we want to implement this method in a statically typed language, we need to answer the question what type should be used for the argument and the return value of this method so that this trait can be used for both Rectangle and VisusalRectangle as well as any other class that has a rectangle characteristics.

Regarding the argument type, the answer is the same as for the method includes: once the language supports a form of multiple subtyping such as interfaces, we declare the argument type to be IRectangle and make sure that all classes supporting the rectangle protocol implement this interface. However, when it comes to the type of the return value, things get more problematic. This is because we would like the method intersect: to return an instance of whatever class it is called on. In particular, this means that an instance of VisualRectangle (Rectangle) must be returned when the method intersect: is called on a RectangleMorph (Rectangle).

What makes this situation difficult is that the return type of the method intersect: is in fact parametric; *i.e.*, it depends on the class to which the trait TRectangle is finally applied. Therefore, using an interface such as IRectangle as the return type does not solve our problem because it would only allow a common subset of all the methods in Rectangle and VisualRectangle to be called on the return value. This problem can be addressed by extending FTJ with a generics mechanism such as that of Generic Java (GJ) [3], recently introduced in Java 1.5. Using generics, we can write the trait TRectangle with a type parameter that is then used as the return type of the method intersect:. And whenever the trait TRectangle is applied to a class such as Rectangle and VisualRectangle, we can then use the corresponding type as the concrete parameter.

CL	::=	`class C<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> ◁ N {`$\overline{\texttt{S}}\ \overline{\texttt{f}}$`;K `$\overline{\texttt{M}}\ \overline{\texttt{TA}}$`}`	*Classes*
TL	::=	`trait T<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> is {`$\overline{\texttt{M}}$`;`$\overline{\texttt{TA}}$`}`	*Traits*
TA	::=	`T<`$\overline{\texttt{S}}$`> | TA with {m@m} | TA minus {m}`	*Trait expressions*
K	::=	`C(`$\overline{\texttt{S}}\ \overline{\texttt{f}}$`) {super(`$\overline{\texttt{f}}$`);this.`$\overline{\texttt{f}}$`=`$\overline{\texttt{f}}$`;}`	*Constructors*
M	::=	`<`$\overline{\texttt{X}}$` ◁ `$\overline{\texttt{N}}$`> S m(`$\overline{\texttt{S}}\ \overline{\texttt{x}}$`) {↑e;}`	*Methods*
e	::=	`x | e.f | e.m<`$\overline{\texttt{S}}$`>(`$\overline{\texttt{e}}$`) | new N(`$\overline{\texttt{e}}$`) | (N)e`	*Expressions*
S	::=	`X |` N	*Types*
N	::=	`C<`$\overline{\texttt{S}}$`>`	*Nonvariable types*

Figure 9: FTGJ Syntax.

$$[\![\texttt{class C<}\overline{\texttt{X}}\texttt{} \triangleleft \texttt{}\overline{\texttt{N}}\texttt{> } \triangleleft \texttt{ N } \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}}\ \overline{\texttt{TA}}\texttt{\}}]\!] \stackrel{\text{def}}{=} \texttt{class C<}\overline{\texttt{X}}\texttt{} \triangleleft \texttt{}\overline{\texttt{N}}\texttt{> } \triangleleft \texttt{ N } \texttt{\{}\overline{\texttt{S}}\ \overline{\texttt{f}}\texttt{;K }\overline{\texttt{M}} \triangleright [\![\overline{\texttt{TA}}]\!]\texttt{\}} \quad (21)$$

$$[\![\overline{\texttt{TA}}]\!] \stackrel{\text{def}}{=} \sum_{\texttt{TA}_i \in \overline{\texttt{TA}}} [\![\texttt{TA}_i]\!] \quad (22)$$

$$[\![\texttt{T<}\overline{\texttt{S}}\texttt{>}]\!] \stackrel{\text{def}}{=} \mathit{local}(\texttt{T}, \overline{\texttt{S}}) \triangleright [\![\mathit{subtraits}(\texttt{T}, \overline{\texttt{S}})]\!] \quad (23)$$

$$[\![\texttt{TA with m@n}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!][n \rightarrow m] \quad (24)$$

$$[\![\texttt{TA minus m}]\!] \stackrel{\text{def}}{=} [\![\texttt{TA}]\!] - \texttt{m} \quad (25)$$

Figure 10: Flattening FTGJ to FGJ

5.1 FGJ and FTGJ

In their paper about FJ, Igarashi *et al.* also present the calculus Featherweight Generic Java (FGJ) [8], an extension of FJ that models Java with generics. Following the augmentation from FJ to FGJ, we now define the new calculus FTGJ, which is an extension of FTJ with generics. We then show how FTGJ can be mapped to FGJ by defining an extended version of the flattening function from FTJ to FJ shown in Figure 4.

The syntax of FTGJ is shown in Figure 9. The metavariable `X` ranges over type variables, `S` ranges over types, and `N` ranges over nonvariable types (types other than type variables). As in FGJ, we write $\overline{\texttt{X}}$ as a shorthand for $\texttt{X}_1, \ldots \texttt{X}_n$ (and similarly for `S` and `N`), and assume sequences of type variables contain no duplicate names. We also allow `C<>`, `T<>`, and `m<>` to be abbreviated as `C`, `T`, and `m`, respectively.

The syntactic extension from FTJ to FTGJ is now analogous to the syntactic extension from FJ to FGJ. In particular, class definitions, trait definitions, and method definitions include generic type parameters.

Once the FTGJ syntax is defined, we can now define the flattening-based translation from FTGJ to FGJ. This translation is shown in Figure 10. Before we go through the details of the definitions, it is important to note that this translation does not perform any type checks. Consequently, this translation produces an FGJ program for *any* FTGJ program; the

$$lookup(\mathtt{m}, \overline{\mathtt{M}}) \stackrel{\text{def}}{=} \begin{cases} M & \text{if } M = \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \bot & \text{otherwise} \end{cases} \tag{26}$$

$$\frac{\mathtt{trait\ T<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ is\ \{}\overline{\mathtt{M}}\mathtt{;}\overline{\mathtt{TA}}\mathtt{\}}}{local(\mathtt{T}, \overline{\mathtt{S}}) = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{M}} \qquad subtraits(\mathtt{T}, \overline{\mathtt{S}}) = [\overline{\mathtt{S}}/\overline{\mathtt{X}}]\overline{\mathtt{TA}}} \tag{27}$$

$$\overline{\mathtt{M}}[\mathtt{n}\rightarrow\mathtt{m}] \stackrel{\text{def}}{=} \begin{cases} (\overline{\mathtt{M}} \setminus lookup(\mathtt{n}, \overline{\mathtt{M}})), conflict(\mathtt{n}) & \text{if } lookup(\mathtt{n}, \overline{\mathtt{M}}) \neq \bot \\ \overline{\mathtt{M}}, \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ n(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} & \text{if } \mathtt{<}\overline{\mathtt{X}} \lhd \overline{\mathtt{N}}\mathtt{>\ S\ m(}\overline{\mathtt{S}}\ \overline{\mathtt{x}}\mathtt{)\ \{}\uparrow\mathtt{e;\}} \in \overline{\mathtt{M}} \\ \overline{\mathtt{M}} & \text{otherwise} \end{cases} \tag{28}$$

$$\text{where } conflict(\mathtt{m}) = \mathtt{Object\ m()\ \{}\uparrow\bot\mathtt{;\}}$$

Figure 11: Adapted composition operators for FTGJ

generated FGJ program may however be invalid due to inconsistent use of types[2]. Because traits are compiled away in the translation, this means in particular that the bounds of the type parameters of traits are not taken into account. This has the effect that all type parameters in trait definitions are actually unbound; a native type system for FTGJ, however, would use these bounds to perform type-checking of generic traits.

A comparison to the translation from FTJ to FJ (see Figure 4) shows that only the cases (21) and (23) are changed. While (21) reflects the extended class definition syntax of FTGJ, the change in (23) was necessary because a trait T that occurs in $\overline{\mathtt{TA}}$ now takes a sequence $\overline{\mathtt{S}}$ of concrete type parameters. This sequence is then passed as a second argument to an extended form of composition operators *local* and *subtraits*.

Figure 11 defines these two operators together with all the other composition operators from Figure 3 that needed to be adapted. The most interesting case is (27), where we extend the rule defining *local* and *subtraits* so that they take two arguments T and $\overline{\mathtt{S}}$, and then replace the formal parameters in T and its subtraits with $\overline{\mathtt{S}}$ before they return, respectively, the methods and the subtraits of T. As in FGJ, replacing the formal type parameters is done using a simultaneous substitution. The other two definitions (26) and (28) are the same as in Figure 3, except that we use the method syntax of FTGJ instead of FTJ.

6 Implementing Traits

Although the flattening property is a critical aspect for the *semantics* of traits, it is not an especially effective way to implement traits, since it quickly leads to code bloat. In this section we provide a brief overview of the strategy used to implement traits in Squeak Smalltalk. We conjecture that this strategy could easily be adapted for statically-typed languages as well.

To add traits to Squeak Smalltalk, we first extended the implementation of classes to

[2]This means that our translation has a character similar to that of C++ templates, which are only type-checked after being instantiated.

include an additional instance variable to represent the composition clause. This variable defines the traits used by the class, as well as any exclusions and aliases. We then introduced a first-class representation for traits, which are essentially stripped-down classes that can define neither state nor a superclass. In particular, this means that each trait is separately compiled and keeps track of its own method dictionary containing the method objects (bytecode) of all the methods implemented by the trait. Note that in this method dictionary, each trait keeps track of both the provided and required methods.

When a class C uses a trait T, the method dictionary of C is extended with an entry for all the methods in T that are not overridden by C. For an alias, we add to the method dictionary a second entry that associates the new name with the aliased method. Since compiled methods in traits do not usually depend on the location where they are used, the bytecode for the method can be shared between the trait that defines the method and all the classes and traits that use it. The only exception are methods that use the keyword `super` because they store an explicit reference to the superclass in their literal frame. When a trait with such methods is applied to a class, these methods are therefore copied with the entry for the superclass changed appropriately. Note that copying of methods containing sends to `super` could be avoided by modifying the virtual machine to compute `super` when needed.

Our implementation never duplicates source code, and duplicates byte code only if it includes sends to `super`, which is relatively rare in trait methods. Because traits are actually represented in the compiled code, they can also be reflected upon. For example, it is possible to check at runtime whether an object is an instance of a class that uses a certain trait T. A program with traits therefore exhibits the same performance as the corresponding single inheritance program in which all the methods provided by traits are implemented directly in the classes that use those traits. There may be a small performance penalty resulting from the use of accessor methods, but such methods are in any case widely used because they improve maintainability. JIT compilers routinely inline accessors, so we feel that requiring their use is entirely justifiable.

7 Related work

Fischer and Reppy have previously presented a type system for traits, but they did not use the framework of FJ [7]. Instead, they introduced an imperative calculus for statically typed traits in Moby (of the ML family). Another important difference to FTJ is that their type system deals with conflicts and excluded methods only in a simplified and limited way.

Traits are a built-in language mechanism of the language Scala [12], a modern multi-paradigm programming language designed to express common programming patterns in a concise, elegant, and type-safe way. The traits adaptation of Scala is particularly relevant as Scala is a statically typed language with a type system similar to those of Java and C#. Therefore, the Scala designers had to tackle many of the problems and trade-offs that we have addressed in this paper. In Scala, traits are modeled as abstract classes that do not encapsulate state, neither in form of variable definitions nor by providing a constructor with parameters. Consequently, each trait, like each class, also defines a type. This is important because it means that in Scala, traits without any concrete methods play the roles of interfaces, and Scala therefore does not have a separate notion of interfaces. Because Scala does not feature the exclusion and alias operators on traits, the subtype relation on traits is defined in a clean and consistent way: a class (or a trait) is always a subtype of all the types

corresponding to the used traits. In summary, we can say that the integration of traits in Scala nicely corresponds to the flattening-based principle proposed in this paper.

Traits-mini-java (TMJ) [13] is an implementation of traits based on a subset of Java. While TMJ does not feature generics (it is based on Java prior to version 1.5), it addresses the problem of typing trait methods by reifying the class that actually uses a trait. This means that TMJ features a new keyword ThisType, which can be used in traits to refer to the class where the trait will eventually be used. Although this is not as expressive as a more general notion of generics (*e.g.*, the one featured in Java 1.5 and Scala), but it has the advantage that often leads to simpler and more concise programs because it does not require an explicit parameter for the class type. Furthermore, the keyword ThisType is, unlike generic type parameters, fully equivalent to the (name of the) class it refers to, and it can therefore be used to create new instance of this class.

Denier and Cointe have introduced traits in Java using AspectJ [5]. Their approach is based on AspectJ introductions and interfaces: the interface declares the method signatures of traits while the introduction defines the behavior (*i.e.* methods) of traits. They also discuss how state can be introduced into the model. Their work is an interesting illustration of the flattening property approach when the run-time of the language cannot be changed to support traits as first class entities as in Squeak or Scala.

In the programming language literature, the term "trait" has been used for a variety of concepts that are related but not identical to the trait construct that is the subject of this paper.

In their theory of objects [1], Abadi and Cardelli use the term trait to denote a collection of methods that is intended as a modular fragment of object behavior. While multiple of those traits may be combined to generate individual objects, this kind of trait combination is significantly different from our notion of trait composition. For example, there is no handling of conflicts, no exclusion, and no alias operation.

8 Concluding remarks

We have shown how the trait flattening property can serve as guideline for a first approach to introduce traits to an existing programming language. We are currently applying this approach to introduce traits to C# in the context of the Rotor Shared Source Common Language Infrastructure. We obtain a rapid prototype of traits for C# by defining traits as a syntactic extension to C#, and then compiling away traits by flattening [14], essentially as described in this paper. Nevertheless C# poses a few additional wrinkles. In particular, in C# only `virtual` methods may be overridden, and one must explicitly declare when one is overriding an inherited method as opposed to defining a new one. Trait composition (and flattening) must take this into account in order to yield correct results.

Although *semantically* traits can be flattened, a proper integration of traits in a given language cannot be achieved by mere syntactic transformation. In our Squeak implementation of traits [10, 15] traits are first-class entities from which classes can be composed. First-class traits enable code reuse. In addition we reuse methods at the level of method dictionaries, by physically sharing common methods among traits and classes, without introducing run-time penalties [17]. Similarly, an extension of a statically typed language with traits should be consistent with flattening, but a robust implementation would require a deeper integration of traits with the host language.

Acknowledgments

We gratefully acknowledge the financial support of Microsoft Research for the project "Traits in C#". We warmly thank Luigi Liquori for his helpful comments and insights. We also thank Arnaud Spiwack and Marcus Denker for reviewing the draft submission.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings OOPSLA'03 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, volume 38, pages 47–64, Oct. 2003.

[3] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the Java programming language. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, pages 183–200. ACM Press, 1998.

[4] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings POPL '90*, San Francisco, Jan. 1990.

[5] S. Denier. Traits programming with AspectJ. In P. Cointe, editor, *Actes de la Première Journée Francophone sur le Développement du Logiciel par Aspects (JFDLPA'04)*, pages 62–78, Paris, France, Sept. 2004. Available at http://www.emn.fr/x-info/obasco/events/jfdlpa04/.

[6] S. Ducasse, N. Schärli, O. Nierstrasz, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *Transactions on Programming Languages and Systems*, 2005. under revision.

[7] K. Fisher and J. Reppy. Statically typed traits. Technical Report TR-2003-13, University of Chicago, Department of Computer Science, Dec. 2003.

[8] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *Proceedings OOPSLA '99, ACM SIGPLAN Notices*, pages 132–146, Nov. 1999.

[9] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, May 2001.

[10] A. Lienhard. Bootstrapping Traits. Master's thesis, University of Bern, Nov. 2004.

[11] L. Liquori and A. Spiwack. Adding multiple inheritance to Featherweight Java. INRIA Sophia Antipolis & ENS Cachan, available at www-sop.inria.fr/mirho/Luigi.Liquori/PAPERS/ftj.pdf, 2004.

[12] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the Scala programming language. Technical Report 64, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland, 2004.

[13] P. J. Quitslund. Java traits — improving opportunities for reuse. Technical Report CSE-04-005, OGI School of Science & Engineering, Beaverton, Oregon, USA, Sept. 2004.

[14] S. Reichhart. A protype of traits for C#. Informatikprojekt, University of Bern, 2005. In preparation.

[15] N. Schärli. *Traits — Composing Classes from Behavioral Building Blocks*. PhD thesis, University of Berne, Feb. 2005.

[16] N. Schärli, S. Ducasse, and O. Nierstrasz. Classes = traits + states + glue (beyond mixins and multiple inheritance). In *Proceedings of the International Workshop on Inheritance*, 2002.

[17] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings ECOOP 2003 (European Conference on Object-Oriented Programming)*, volume 2743 of *LNCS*, pages 248–274. Springer Verlag, July 2003.